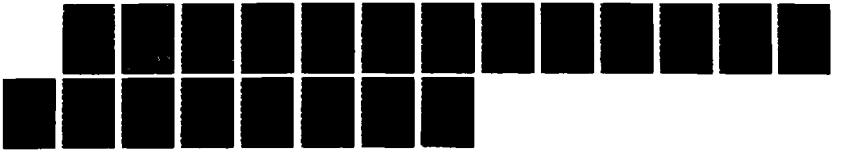
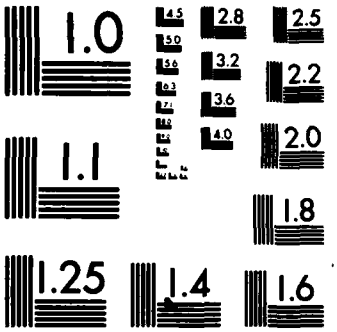
AD-A171 039 A REWRITE RULE MACHINE SIMULATION OF CONCURRENT TREE REWRITING(U) SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB T WINKLER ET AL 10 JUL 86 1/1

UNCLASSIFIED N00014-85-C-0417 F/G 9/2 NL





SRI International

A REWRITE RULE MACHINE

Simulation of Concurrent Tree Rewriting

Final Report
July 1986

By: Timothy Winkler, Computer Scientist
Sany Leinwand, Senior Research Engineer
Joseph Goguen, Program Manager

SRI Project ECU 1243

Prepared for:

Office of Naval Research
Information Sciences Division
800 N. Quincy St.
Arlington, VA 22217-5000

Attn: Dr. Charles Holland, Code 1133

Contract No. N00014-85-C-0417

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
(415) 326-6200
TWX: 9100-373-2046
Telex: 334486

Simulation of Concurrent Tree Rewriting*

Timothy C. Winkler, Sany Leinwand, and Joseph A. Goguen
SRI International, Computer Science Lab

July 10, 1986

Abstract: This report presents results of simulated concurrent tree rewriting of programs in the OBJ2 language, as well as descriptive statistics for the rule sets of typical OBJ2 examples. This work explores the use of OBJ2 as a concurrent programming language, explores in detail the implications of the concurrent tree rewriting model for realistic problems, and provides information for design decisions about the Rewrite Rule Machine. We found that it was easy to write good concurrent programs in OBJ2; most rewrite rules from the OBJ2 examples are simple and fall into easy-to-handle categories. The simulations indicate that concurrently executed interpreted programs will often be faster than sequentially executed compiled programs. This report also presents some techniques for controlling tree size in concurrent rewriting. Finally, we present some high-level simulation results for a VLSI rewrite rule engine that we are considering implementing. The results are highly favorable.

1 Introduction

This report is part of the Rewrite Rule Machine (RRM) project at SRI International. Its results support the feasibility and desirability of using an ultra-high-level language (UHLL) like OBJ2 [1] to program the RRM. It also considers some questions that arise in designing the RRM that are difficult to answer on theoretical grounds, or that have to do with what can be expected in practice. Such problems can be attacked with simulations and by examining properties of typical programs. The following sections discuss the sample programs, report the simulated performance of these programs, and consider the concurrent execution of interpreted programs. We then investigate strategies for rewriting, examine the size of the tree in computations of these programs, and study the behavior of these programs in a high level model of a proposed design for a processing node of the RRM.

1.1 Goals

We wish to answer a number of practical questions about concurrent tree rewriting, including:

1. How easy is it to program the RRM with OBJ2?

*Supported by Office of Naval Research Contract N00014-85-C-0417.

2. How effectively does OBJ2 capture the concurrency that exists in typical problems?

3. How complex are typical rules and rule sets?

4. How much space do typical concurrent programs consume?

5. In the single instruction stream/multiple data stream (SIMD) design, rules will be performed one at a time. How does this compare with fully concurrent rewriting?

2 Review of OBJ2

OBJ2 [1] is a functional programming language with an operational semantics given by tree rewriting and a mathematical (or "denotational") semantics given by equational logic. Thus, OBJ2 is a declarative language, since its statements have a declarative reading as equations stating properties that one desires the solution to have; in effect, they describe the *problem* to be solved. Moreover, OBJ2 is particularly suitable for the RRM because its operational semantics is tree rewriting. In addition, OBJ2 has a very expressive and flexible type system, including overloading and subtypes; OBJ2 also has user-definable abstract data types (with user-definable "mixfix" syntax) and perhaps the most powerful generic module mechanism available in any current programming language. Moreover, OBJ2 is a "wide spectrum" language that elegantly integrates coding, specification, design, and verification into a single framework.

The following simple program illustrates some basic features of OBJ2:

```
obj BITS is
  protecting INT .
  sorts Bit Bits
  ops 0 1 : -> Bit .
  op nil : -> Bits .
  op _._ : Bit Bits -> Bits .
  op length_ : Bits -> Int .
  var B : Bit
  var S : Bits
  eq : length nil = 0 .
  eq : length B . S = inc length S .
endo
```

OBJ2's basic entity is the *object*, a module encapsulating executable code. The keywords `obj ... endo` delimit the text of the object. Immediately after the initial keyword `obj` comes the object name, in this case `BITS`; then comes a declaration indicating that the built-in object `NAT` is imported. This is followed by declarations of new data sorts, in this case `Bit` and `Bits`, and of the constants `0 1 nil`, and the operations `_._` and `length_`, each with information about the distribution and sorts of arguments and the sort of the result; underbar characters "`_`" are used to indicate argument places for mixfix operators; thus `_._` is infix and `length_` is prefix. Finally, variables of sorts `Bit` and `Bits` are declared and two equations constituting the body

of the object are given; inc is the increment (i.e., add 1) function on integers. Trees (i.e., terms) are built up from the constants and function symbols. For example, two such terms are `3 . (7 . nil)` and `length 1 . (8 . (3 . nil))`; the latter evaluates to 3, by applying the two equations as left-to-right rewrite rules. The operator `_._` is called a *constructor* since it has no equations.

The OBJ2 code is executed using a set of rules derived from the equations given in the object definitions by a compilation process. At any point, the current state of the computation is a term that can be viewed as a labelled tree. A "rewrite rule" consists of a lefthand side (or LHS) which is a pattern, a righthand side (or RHS) which is a replacement template, and possibly a condition, each of which is a term usually containing variables. A rewrite rule is applied to a tree by matching the LHS against some portion of the tree, remembering what the variables in the pattern match against, testing whether the condition as a logical expression holds for those values of the variables, and if this is so, replacing the subtree matched by the result of instantiating the RHS template using the values of the variables.

Certain "built-in" rules, used for efficiency, describe operations such as addition for natural numbers. These rules still have LHSs, but the RHSs may be described in some more operational form.

3 Simulating Concurrent Rewriting

The generalization of tree rewriting to the concurrent case is discussed in [2]. In general, there are many rewrites that can be performed on a tree. In the sequential case, you choose which to do first; in the concurrent case setting, you choose a set to do simultaneously.

The concurrent tree rewriting simulator was constructed by modifying the existing sequential OBJ2 interpreter to do concurrent rewriting by using a top-down maximal strategy for choosing a nonoverlapping set of rewrites. At a given instant, two rules may match overlapping portions of the tree, specifying two different ways that this portion of the tree could be altered. In such a situation, only one rule can be used, and a strategy is needed for choosing which rule to use. The top-down strategy gives preference to the rule that matches at a higher location. Furthermore, as many reductions as possible are done, so that the strategy is maximal.

The current state of the simulated computation is always a single term (or tree). A single simulated concurrent tree rewrite step consists of

1. Labelling the tree with all possible rewrites,
2. Choosing a nonoverlapping set of rewrites,
3. Performing this set of rewrites

Statistics, such as the size of the tree at each point, are accumulated as the simulation proceeds. This simulation process was relatively easy to implement, given the facilities provided by the current OBJ2 implementation.

4 The Examples Considered

The examples considered in greatest detail in our simulation studies were calculating the n^{th} Fibonacci number, matrix multiplication, and sorting. Many other examples have also been done, including the prime numbers generated in a "lazy stream" and an interpreter for a higher-order functional language. These examples cover a wide range of basic problems: for example, Fast Fourier Transform (FFT) can be viewed as matrix multiplication, and results obtained for matrix multiplication also give good results for FFT; moreover, communication routing problems can be viewed as sorting problems. The interpreter is typical of many software development tools. The selection of problems was influenced by looking at the literature on concurrent algorithms.

We first wrote very straightforward programs for the Fibonacci numbers, matrix multiplication, and sorting problems in OBJ2. For the Fibonacci problem, we used the straightforward recursive definition; for matrix multiplication, we used lists to represent matrices (thus, a matrix is a list of rows which are lists of matrix elements, and the product matrix is the list of products of rows of the left matrix by the right matrix); for the sorting solution we used a simple merge sort. These programs were written without any concern for possible concurrency.

In order to investigate how much concurrency could be exploited by OBJ2, the matrix multiplication and sorting problems were rewritten to use tree-structured data rather than lists, which naturally gives rise to "divide-and-conquer" solutions. The tree-structured versions are only about 50than the straightforward list versions. However, especially in the case of sorting, they are conceptually more difficult; the sorting version is essentially a bitonic merge sort.

4.1 Structural Properties of the Examples

Many properties of the rule sets that correspond to OBJ2 programs are interesting, including the size of the LHS and RHS of rules, number of rules in a module, and properties of the rules, such as left-linearity and conditionality. We have studied the properties of all the primary simulation examples and, in addition, have looked at examples derived from the "standard prelude" of OBJ2.

4.2 Results

We can sketch the structure of a typical OBJ2 module:

1. It usually introduces one or two new sorts.
2. It has about five operators.
3. It has about six equations.
4. There is a very good chance that it has no conditional equations.
5. There is a very good chance that all equations are left-linear (this will be discussed below).
6. There is a very good chance that there is no possible overlap of the matches of rules.

This description summarizes the properties of the key sample problems. What is important here is that modules are relatively small and simple. It is expected that they will be a useful way to view user programs in the compiler. For example, one problem in the RRM is distributing rules. Since modules are small and consist of logically related parts of the program, the process of distributing rules might treat modules as basic units.

Global statistics on the number of argument positions of operators were gathered. The average number of arguments was 1.28, and the maximum number was 5. Of the operators, roughly 60% were unary and 38account for roughly 98% of all operators. These statistics imply that it will be typical for tree size to be approximated by $2^{d+1} - 1$, where d is the depth of the tree, since this is the maximum number of nodes in a tree with depth d having only binary branches.

The most important view of an OBJ2 program, for the rewriting process, is as a rule set. We collected global statistics on the structure of rule sets arising in the examples.

In general, rules are relatively simple. Two statistics were collected: size and depth. The size of a term (for example the LHS of a rule) is the total number of nodes in the tree representing it. The depth of a term is the height of the tree that represents it. The average size of the LHS of a rule, over the sample programs, is 4.3 nodes (the maximum was 9); at least 80% of the rules had a LHS with size 5 or smaller. This and similar statistics covering depth, RHS and conditions for conditional rules are summarized in the following table:

What	Average	Maximum	80% level
LHS size	4.3	9	5
LHS depth	2.8	5	3
RHS size	3.9	24	6
RHS depth	2.3	7	3
Cond. size	4.7	8	8
Cond. depth	3.0	5	5

The most important comment about conditional rules is that they are fairly rare, less than 3% in the example programs. We can see from these data that, in practice, rewrite rules tend to be simple. This information may also be used to make lower-bound estimates on the amount of information needed to represent a typical rule set. The number of bits needed to represent a rule set will be proportional to the combined number of nodes in the LHS and RHS of the rule, which will be about eight or nine. Since a typical number of rules is six, a rule set for a module will contain about fifty references to operator names. In addition, the time required for pattern matching is generally dependent on the depth of the pattern. The data above indicate that, in the typical case, matching the LHS of a rule will be a fast operation.

Additional properties of rules that are important are left- and right-linearity; failure of either of these properties to hold results in some complexity in the implementation. By far the most important is left-linearity, which is the property that the LHS of the rule has at most one occurrence of each variable. If there are two occurrences, the matching process, in order to succeed, must compare two whole subtrees for equality. In general this is not a "local" operation and will be expensive. It

turns out that none of the rules in the sample are non-left-linear. Right-linearity is analogous; each variable *occurs at most once* in the RHS. About 17% of the rules were non-right-linear. In practice, non-right-linearity might require that portions of the tree being rewritten be copied.

A very important global property of sets of rules is whether or not there are possible overlaps between the matches of rules. If two matches of rules overlap, the overlap must be detected and only one of the rules be allowed to proceed and perform a replacement on the tree. If none of the operators below the top operator in the LHS of a rule, has any rules, i.e. the operators are all constructors, then no overlap is possible. In the sample considered, 78% of the rules satisfied this condition; there are no possible overlaps with their matches. It is quite natural for all of the operators below the top of the LHS of a rule to be constructors. A natural programming style uses recursive definitions on structures given by constructors, which results in rules of this kind.

Many possibilities allowed by the above test don't result in true overlaps. In fact, only 18% of the rules might involve overlaps. The true overlaps are all found in the matrix multiplication examples where the programs are viewed as operating on "quarter planes" rather than rectangular matrices. The rules that cause the problem are certain rules that reduce "larger" planes to simpler ones that are equivalent–for example, by deleting empty rows. If these rules are omitted, we obtain slightly different versions of the matrix multiplication programs that compute the same results for square matrices whose entries are not zero. Furthermore, these rules are applied only at the bottom of the tree.

It seems very likely, based on these results for overlap, that the primary mode of computation might assume no overlap and that the overlapping cases might receive special treatment, perhaps with lower efficiency.

4.3 Discussion

We have found that the rule sets that arise in practice are relatively simple and straightforward to handle. Many complex or problematic cases, such as non-left-linear rules, do not arise in practice or are rare.

Further, we have supported the views that modules tend to be small, and that the typical rewrite rule is unconditional, left-linear, and does not overlap with other rules.

5 Evaluating an UHLL for Concurrent Programming

We have done simulations at the level of concurrent tree rewriting, using the modified OBJ2 interpreter. We discovered that it was not difficult to write highly abstract OBJ2 programs that effectively captured the concurrency that existed in the problems considered.

5.1 The Results

First we studied the natural and straightforward solutions to the Fibonacci, matrix multiplication, and sorting problems. The time taken to compute the answer concurrently was compared against the number of steps required in the current OBJ2

interpreter (and against general theoretical results). The current interpreter is a good sequential implementation of OBJ2 using a number of clever optimizations, so that it presents a good estimate of the time needed for a sequential implementation of these programs. In each case there was a significant speedup, as seen by the ratio of the sequential time to the concurrent time. The results were:

Problem	Sequential	Concurrent	Ratio
Fibo	a^n	n	exponential
mm	n^3	n	n^2
sort	$n \log n$	n	$\log n$

where a is the golden mean, about 1.618. These results strongly support our claim that natural and straightforward OBJ2 programs often exploit much of the concurrency available in a problem.

As might be expected, when a Fibonacci algorithm that runs in logarithmic sequential time was coded in OBJ2, its execution by concurrent tree rewriting was only slightly faster than the sequential computation. It is worth recalling that some problems are *inherently sequential* in the sense that there is a sequential solution which no concurrent solution can outperform.

We investigated how much concurrency could be exploited by OBJ2. The "divide-and-conquer" tree-structured solutions to the matrix multiplication and sorting problems were simulated. The results were as follows, where "Max Term Size" entries refer to the concurrent case:

Problem	Sequential	Concurrent	Speedup	Max Term Size
mm	n^3	$\log n$	exponential	n^3
sort	$n(\log n)^2$	$(\log n)^2$	n	n^2

The matrix multiply solution in terms of trees is a natural generalization of the list solution when the tree formation operation is viewed as concatenation on "structured" lists, and was easy to write. The tree solution for sorting was more difficult. The basic algorithm is a classical one due to Batcher, the bitonic merge sort (see [3]), where one of the key operations is the "shuffling" of two sequences. Expressing the shuffling operation in terms of trees was not completely straightforward. This basic operation of shuffling two sequences can be expressed in OBJ2 as follows:

```
shuffle(<x>) = <x> .
shuffle(s1 ^ s2) = pairup(s1,s2) .
pairup(<x>,<y>) = <x> ^ <y> .
pairup(s1 ^ s2, s1 ^ s2) = pairup(s1,s1) ^ pairup(s2,s2) .
```

where ^ is the tree formation operator and leaves have the form <x>. The operator pairup actually does the work of shuffling two sequences. The third equation above says that, to shuffle two sequences, one should split each into two parts, shuffle the corresponding pieces, and then combine the shuffled results.

If we compare these results with theoretical results for VLSI circuits, as in [5] and [6], taking the area of the circuit to be the maximum size of a tree arising in the concurrent simulation, we have the following (the lower bounds come from the AT^2 formula in [5] and [6]):

Problem	Concurrent	AT^2 Lower Bound
mm	$n^3(\log n)^2$	n^4
sort	$n^2(\log n)^4$	n^2

Sorting achieves performance very close to the lower bound (which is typical for realistic VLSI sorting designs). Matrix multiplication shows a simulated performance that is not realizable in practice, i.e., no actual VLSI implementation could attain this performance, because of its space requirements. The matrix multiplication algorithm essentially takes two $n \times n$ matrices and forms a matrix of pairs, each pair consisting of a row from one matrix and a column from the other, and then applies an inner-product operation to each of these pairs. Each row or column is duplicated n times. In VLSI designs, this rearrangement and duplication of data requires a large area.

The maximum tree size arising in computing the Fibonacci numbers with simulated concurrent tree rewriting grew very rapidly with n; but any real machine will have only a limited capacity to represent terms. This issue is discussed further in Section 6.

5.2 Concurrency and Interpreters

In another experiment, we translated the tree solution of the matrix multiplication problem into a higher-order functional language (a variant of Backus' FP language) defined within OBJ2. The definition of the higher-order language within OBJ2 gives an interpreter for that language. Simulations of the tree matrix multiplication program for this "language within a language" were done, and the results indicate that interpreted programs, concurrently executed, may be significantly faster than compiled solutions executed sequentially. Interpretive approaches are attractive because they are easier to develop than compiled approaches.

The exact comparison made was between the number of concurrent tree rewriting steps required for the interpreted higher-order functional version of tree matrix multiplication with the number of rewriting steps required by the current OBJ2 interpreter for the original noninterpreted, nontree version. We are using a single rewriting step as a unit of measurement. In fact, this favors the sequential version since the specialized hardware of the RRM will perform rewriting steps more quickly than conventional hardware.

The concurrently executed interpreted matrix multiplication is probably several times faster than the sequentially executed noninterpreted matrix multiplication for matrices as small as 7×7. The time required by the interpreted version is $39 \log n$ steps compared with $5 \log n$ steps for the original concurrently executed noninterpreted tree version. (This is consistent with what is known for conventional systems, where interpreted programs typically are about eight times slower than direct versions.) The time required for the sequential version of this problem is about n^3 (as discussed previously). The advantages of concurrency outweigh the overhead of interpretation.

5.3 Discussion

A major finding is that it is very easy and natural to express algorithms in OBJ2, in such a way as to take advantage of most of the inherent concurrency in the problem,

and thus to run surprisingly fast on the RRM.

An important observation arising from our work on simulation is that, although it is not difficult to get essentially time optimal performance from OBJ2 programs, it is possible for these programs to consume great amounts of space. Comparing the concurrently executed higher-order functional matrix multiplication program with the sequentially executed version indicates that this concurrently executed interpreted program would be faster, in many cases, than a compiled version on a conventional machine. If the RRM is used as a development environment for programs intended for a conventional machine, then interpreters sometimes can be used without a critical loss of efficiency (the performance may in fact be better than that on a conventional machine).

6 On the Complexity of Rewriting Strategies

The time taken to compute a result in concurrent tree rewriting may be strongly influenced by the strategy used for selecting a nonoverlapping set of rewrites. The "top-down maximal" strategy for choosing a nonoverlapping set of rewrites was implemented and used in all the examples discussed in the Section 5. It was originally conjectured that this would be a good strategy in general, but this is not the case.

6.1 The Problem of Optimality

Detailed examination of a particular test case for strategies for choosing a nonoverlapping set, for which the top-down strategy gives a poor result, has convinced us that the problem is difficult in general. It fact, it appears clear that there are cases where any "maximal" strategy is a bad one, and furthermore, there is no easy way to improve the top-down maximal strategy.

One of the simplest test cases for a nonoverlapping set choice strategy is the single rule:

```
(x * y) * z = x * (y * z)
```

This is the associative law taken as a rule, and when applied to a tree eventually places it in a fully "right-associated" form, i.e. a single long branch down the right. Assume there are no other rules for "*".

If one uses the top-down maximal strategy in applying this rule to a linear branch of length n down the left, it takes about $n/2$ steps to rewrite it to a linear branch down the right (which can be rewritten no further). This result was observed in simulations of this rewriting problem.

However, there is a strategy that can do this rewriting task in $2 \log n$ steps. In outline, it is this: first perform reductions using as many applications as possible of the rule shown above for matches of the rule somewhere along the leftmost edge of the tree. Eventually the tree will be reduced to just a few nodes along that edge. At the mid-point, and only then, apply the rule just a few times at the top node. Then enter a second phase where the rule is applied only on the rightmost edge of the tree. (The trees arising in the second phase are just the reversal of the trees arising in the first phase.) This strategy has the property that many possible reductions in the middle part of the tree are ignored during the rewriting, so the strategy is

nonmaximal. Note further that knowing whether or not to apply the rule requires global information about the context. This decision cannot be based solely on a limited portion of the tree, since patterns in the middle portion of the tree would be confused with the patterns on the left (or right) edge of the tree. The correctness and time required for this strategy have been verified by simulation.

A strategy that is maximal will perform reductions in the part of the tree below the left edge; in general, these reductions create configurations that take a long time to transform into the final form.

6.2 Discussion

It appears that it will be difficult and costly to design and implement an optimal strategy for the choice of nonoverlapping sets of reductions. However, as the structural study of the rules corresponding to the sample problems indicates, the strategy for selecting non-overlapping sets of reductions is not critical since overlapping matches are relatively rare. The conclusions of this section, therefore, should not be seen as discouraging. Furthermore, in the low-level SIMD design for the RRM, which will perform reductions one rule at a time, only overlaps of rules with themselves will be difficult. Problems of this kind did not arise in any of the sample programs that were simulated.

7 Tree Size in Concurrent Rewriting

In this section, we discuss the problem of the tremendous growth in the size of the tree seen in some examples, focusing especially on those cases when such growth is not productive. The primary goal of a concurrent machine is speed; on the other hand, it will have limited resources for representing trees and we have seen that the tree size can get very large in natural examples.

The large growth of tree size in a concurrent computation is not always a problem. In general, it is the basis for the speedup obtained from concurrency. The speedup can never be larger than the maximum tree size because operations take place at locations in the tree and at any moment only as many operations can take place as there are places in the tree. For example, the simplest solution to the Fibonacci problem shows an exponential growth in the size of the tree, but corresponding speedup prevents this from being a problem. The space used by nearly all of the programs seems to be productive.

Even when the use of a great deal of space is productive, it will be necessary to consider means of controlling the growth, since the resources of the machine may be limited. In particular, control will be necessary when the growth is very rapid as a function of the problem size. The same techniques can be used to control both productive and nonproductive growth.

7.1 Observations on Tree Size Growth

Certain natural definitions result in a very rapid growth of tree size that is not essential to the problem. For example, in a logarithmic time solution to the Fibonacci problem the time taken with concurrent tree rewriting was roughly the same as the time taken in a sequential interpreter. In the sequential interpretation (which used a

bottom-up strategy) the space required to compute the n^{th} Fibonacci number grew as $\log n$. In a concurrent rewriting computation, we consider the space required to be the maximum size of a tree arising in the computation. In the concurrent execution of this Fibonacci program, the space required grew roughly as $1000n$. This was due to a number of non-right-linear rules, which duplicate subtrees. For example, the rule

```
sqr(p) = prod(p,p)
```

creates two copies of that portion of the tree matched by p. A rule like this, when applied repeatedly, can tremendously expand tree size. This rule was replaced by the specialization

```
sqr(<< a ; b >>) = prod(<< a ; b >>,<< a ; b >>)
```

which is derived by replacing the variable p with the more specific pattern `<< a ; b >>` in the original rule. Similar changes were made to some of the other rules, and the space needed was reduced to approximately $40n$.

To a certain extent, the specialized rule requires a bottom-up order of evaluation. It will not be applied until its argument is evaluated further than is required for application of the unspecialized rule. Because the evaluated form of the argument will be smaller than the unevaluated form of the argument, less tree structure will be duplicated. The remaining rules that involve repeated variables in their RHS have variables that matched numeric quantities so the process of specialization cannot be used. If something equivalent is done, which is to make these rules conditional with a special condition checking that the value of variables are numbers (not expressions), then the space requirements will be the same as for the sequential interpreter. All of these transformations are straightforward, and could even be done mechanically.

Another strategy for improving the rate of term growth is to change unconditional equations that use `if_then_else_fi` to a pair of equations that are conditional, provided the condition is very simple, since conditional equations with certain kinds of very simple conditions will be applied rapidly on the RRM. For example, the equation

```
fibo(n) = if n < 2 then n else fibo(n-1) + fibo(n-2) fi
```

might be transformed into the two conditional equations

```
fibo(n) = n if n < 2
fibo(n) = fibo(n-1) + fibo(n-2) if 2 <= n
```

This transformation prevents the creation of duplications between the condition in the `if_then_else_fi` and the `then` and `else` branches. The subtree matched by `n` will be duplicated four times when the unconditional rule is applied since `n` occurs four times in the RHS of that rule.

In addition, we have considered ways that the E-strategies of OBJ2 (discussed in [1]) can be generalized to the concurrent setting. This means that sometimes the set of nonoverlapping reductions will not be maximal. If speed were the only consideration, we would always choose a maximal set. This subject is discussed in the report [2].

We have considered methods to control tree size in concurrent rewriting. The techniques considered for solving this problem are: redesign the algorithm (undesirable in general), use generalized E-strategies, or tune the given algorithm by specializing rules and using conditional rules.

7.2 Data Representation for Terms

The choice of representation of terms can have a big effect on the amount of space needed during a computation. During a concurrent computation, certain portions of the data will be used by subcomputations taking place at different locations. When this is the case, we may either duplicate (i.e., copy) the data at each place where it is to be used, or keep it in one place and provide some means for shared access. The latter will result in saving space: the resources used for representing the tree will be smaller. In other words, instead of representing the state of the rewriting process as a tree, we might choose to represent it as a directed acyclic graph (dag). The repeated use of a subtree naturally arises when there is a variable that occurs in two places in the RHS of a rule. (In the results given above, the tree size was always given with the assumption that the representation was a tree.)

The choice of trees versus dags is not clear-cut; each has advantages and disadvantages. For trees, when a computation is started that uses some data repeatedly, the cost must be paid of copying the data (both in time and space). For dags, when a computation is started that uses some data repeatedly, there is no immediate cost, but there may be later because of delays due to shared access (access may have to be sequentialized, and access to a remote location may involve transit delays). Also, in general the existence of sharing results in more complex control for the processes operating on the data.

The degree of compression obtainable by using a dag instead of a tree can be arbitrarily large. For example it is possible to represent a full binary tree with height n, all leaves identical, using space n rather than $2^{n+1} - 1$.

We have created a version of the concurrent tree rewriting simulator that works with dags and have also instrumented the basic simulator to keep track of the amount of work done in copying. The results are summarized in the following tables:

Program	Instance	Max Size	dag size	Ratio
mm	8 × 8	4431	997	4.4
sort	length 32	1482	231	6.4
tree mm	8 × 8	7111	1287	5.5
tree sort	length 32	494	382	1.3

The columns of the table indicate the program, size of the sample problem, the maximum tree size arising in a strict tree approach, the maximum tree size for dags, and the ratio of these last two values. These examples make it is clear that using dags can result in a large saving of space. Consider the following table of data on copying:

Program	Instance	Copying	Concurrent steps	Per step	Percentage
mm	8×8	7349	34	216	4.9%
sort	length 32	10847	82	132	1.9%
tree mm	8×8	9534	20	476	8.9%
tree sort	length 32	11305	91	124	25.1%

The columns, from the third onward, give the total number of tree nodes copied in doing the example reduction, the number of concurrent steps, the average number of nodes copied per step, and finally the percentage that the average (from the previous column) is of the maximum size (given in the previous table). The value for number of nodes copied is the cumulative total number of nodes copied over all rewriting steps. From this we can see that the portion of the tree that needs to be copied is, on the average, relatively small. The exception is the last case, where about one quarter of the tree, on the average, is copied in each step. This seems problematic.

It appears that there are compelling reasons to consider using a dag representation for the tree, at least at some level (see for instance the discussion in [2]). The final design might be a hybrid system with trees within processing nodes and a dag structure overall. However, the issue of tree versus dag will probably be resolved only at lower, more detailed, levels of modelling.

8 SIMD Processing Node Model

The low-level RRM design being considered will operate in a single instruction stream/multiple data stream (SIMD) mode, in which there is a single control unit for a large number of very simple rewriting cells (see [4] for a discussion of the RRM architecture). The single controller will be able to direct all cells to do one basic activity at a time, perhaps with some local variations. For example, it could direct all cells to simultaneously attempt to apply a certain rewrite rule.

We can model this design at a very high level by performing the simulation as before, but, in addition, keeping statistics on the number of distinct rules applied in a single concurrent step. If we assume that all possible reductions by a single rule can be done simultaneously, but that different rules must be done sequentially, then we can roughly model the SIMD performance by treating each concurrent step as consisting of some number of "single-rule" steps. The SIMD model will thus always require more steps, but may be desirable because it is more economical to share a single controller. In fact, a single SIMD step might take less time than a step on a more complex machine that would allow arbitrary concurrent tree rewriting.

It is important in comparing the general and SIMD cases that the strategy for rewriting be the same, since variations in strategy could determine the results. This is ensured by the above simulation process.

8.1 Results

The two matrix multiplication examples were examined in this light. For 8×8 matrices we have

Version	Concurrent	Single-Rule	Average steps	Maximum steps
mm	34	166	4.9	8
tree mm	20	39	2.0	6

The columns of the tabulation indicate the number of concurrent tree rewriting steps required, the number of single-rule steps required (as discussed above), the average number of single-rule steps per concurrent step, and the maximum number of single-rule steps for any concurrent step.

The tree version is very favorable to the SIMD notion since 11 of the 20 steps involve only a single rule, and all of the multiplications are done in a single concurrent step (512 operations). This results because the computation in the example proceeds as a number of globally synchronized phases. First the rows and columns of the matrices are distributed as needed. Because of the symmetry of the structures, all of the independent parallel processes involved in doing this finish at the same time; then all of the inner products are started. The inner products also proceed symmetrically so that the multiplications are all done at once. The computation finishes by summing up the results. Each of the phases involves only a small subset of the rules (perhaps a single rule) so that the average number of rules per concurrent step is quite small.

The basic matrix multiplication is probably more typical, and it also appears favorable to the notion of a SIMD design. Here the computation also proceeds as a sequence of phases, but they are not synchronised; in different parts of the overall computation, the phases start at different times. For example, the multiplications are spread over 22 concurrent tree rewriting steps. However, on a larger level, the computation is still proceeding as phases which use only part of the rule set.

The results of other simulations are summarised in the following table:

Program	Concurrent	Single-Rule	Ratio
mm	$4n$	$20n$	5
sort	$3n$	$9n$	3
tree mm	$5 \log n$	$15 \log n$	3
tree sort	$3(\log n)^3$	$6(\log n)^3$	2

In general the number of single-rule steps is less than five times the number of concurrent steps and is often much less than that. The number of distinct rules in a single concurrent step is clearly limited by the size of the given set of rules. In fact, on the average, it seems to be just a small percentage, perhaps 10-20% of the total number of rules.

8.2 Discussion

The simulations suggest that the "single-rule" (SIMD) number of steps required for a given concurrent computation is a small multiple (dependent on the given set of rules and independent of the problem size) of the number of concurrent tree rewriting steps required without this restriction. The use of a single controller should result in a large economy in the design of the rewriting chip, and this will more than compensate for a slight inefficiency resulting from the need to perform reductions a single rule at a time.

9 Conclusions

A major finding is that it is very easy and natural to express algorithms in OBJ2, in such a way as to take advantage of most of the inherent concurrency in the problem, and thus to run surprisingly fast on the RRM. We also found that the advantages of concurrency outweigh the overhead of interpretation. A concurrently executed interpreted program may be faster than a sequentially executed compiled program.

The study of OBJ2 modules shows that they are relatively small and that rules are generally simple and fall into easy-to-handle categories.

Issues related to the size of the tree in the concurrent tree rewriting process have also been investigated by the simulations. Various techniques for reducing the size have been tested and these have been found promising. The issue of tree versus dag in the representation of the tree being rewritten has also been explored. The evidence at this level indicates the superiority of dags, although it is premature to make a decision, because lower-level considerations may predominate.

A very high level simulation of the SIMD design has been performed, with results that are favorable to this design. The approach of applying a single rule at a time over a portion of the tree seems to be very effective because computations tend to proceed in global phases.

Appendix. Some Example Programs

This appendix contains the OBJ2 code for the the basic versions of some of the example programs. We want to stress the simplicity of these particular programs, which, nevertheless, obtain *reasonable performance as concurrent programs.*

A.1 Fibonacci Numbers

The program for this problem is the simplest recursive version, yet it takes tremendous advantage of concurrency.

```
obj FIBO is
  protecting INT .

  op fibo_ : Int -> Int .

  var x : Int .
  eq Int : fibo x = if x < 2 then x
                    else fibo(x - 1) + fibo(x - 2) fi .
endo
```

A.2 Matrix Multiplication

The simple matrix multiplication program operates on matrices represented as lists of rows, which are represented as lists of numbers. These representations are introduced by the modules ROW-INT and MATRIX-INT. The matrix multiplication operator *m is defined in terms of tr, which transposes a matrix, and ip, which is

inner-product. To multiply two matrices, first the right matrix argument is transposed, then inner product is applied to rows of the left matrix and columns of the right matrix (the rows of the right matrix transposed), as follows:

```
obj LIST[X :: TRIV] is
  sort List .

  op empty : -> List .
  op _al_ : Elt List -> List .
  op hd_ : List -> Elt .
  op tl_ : List -> List .

  var e : Elt .
  var l : List .
  eq : hd (e al l) = e .
  eq : tl (e al l) = l .
endo

obj ROW-INT is
  protecting LIST[INT] * (sort List to Row,
                          op (empty) to (empty-row)) .
  eq : 0 al empty-row = empty-row .
endo

obj MATRIX-INT is
  protecting LIST[ROW-INT] * (sort List to Matrix,
                              op (empty) to (empty-matrix)) .
  eq : empty-row al empty-matrix = empty-matrix .
endo

obj MM is
  protecting MATRIX-INT .

  op _*m_ : Matrix Matrix -> Matrix .
  op mmtr : Matrix Matrix -> Matrix .
  op rowxtr : Row Matrix -> Row .

  op _ip_ : Row Row -> Int .

  op tr_ : Matrix -> Matrix .
  op firsts_ : Matrix -> Row .
  op rests_ : Matrix -> Matrix .

  var x y i : Int .
  var M N : Matrix .
  var A B R : Row .

  eq : M *m N = mmtr(M, tr N) .
```

```
  eq : mmtr(empty-matrix,N) = empty-matrix .
  eq : mmtr(A al M, N) = rowxtr(A,N) al mmtr(M,N) .

  eq : rowxtr(A,empty-matrix) = empty-row .
  eq : rowxtr(A,B al N) = (A ip B) al rowxtr(A,N) .

  eq : empty-row ip A = 0 .
  eq : A ip empty-row = 0 .
  eq : (x al A) ip (y al B) = (x * y) + (A ip B) .

  eq : tr empty-matrix = empty-matrix .
  ceq : tr M = (firsts M) al (tr (rests M))
               if M =/= empty-matrix .

  eq : firsts empty-matrix = empty-row .
  eq : firsts (empty-row al M) = 0 al (firsts M) .
  eq : firsts ((i al R) al M) = i al (firsts M) .

  eq : rests empty-matrix = empty-matrix .
  eq : rests (empty-row al M) = empty-row al (rests M) .
  eq : rests ((i al R) al M) = R al (rests M) .
endo
```

A.3 Merge Sort

The merge sort sorts a sequence by splitting it into two halves (the even and the odd numbered subsequences), sorting these halves, and then merging the results together:

```
obj SORTER is
  protecting LIST[INT] .

  op merge-sort  : List -> List .
  op odds_ : List -> List .
  op evens_ : List -> List .
  op merge : List List -> List .

  var x y : Int .
  var L M : List .

  eq : merge-sort(empty) = empty .
  eq : merge-sort(x al empty) = x al empty .
  eq : merge-sort(x al (y al L)) =
     merge(merge-sort (x al (odds L)),
           merge-sort (y al (evens L))) .

  eq : evens empty = empty .
```

```
  eq : odds empty = empty .
  eq : odds (x al L) = x al (evens L) .
  eq : evens (x al L) = (odds L) .

  eq : merge(x al L, y al M) =
    if x < y then x al merge(L, y al M)
    else y al merge(x al L, M) fi .
  eq : merge(empty, M) = M .
  eq : merge(L, empty) = L .
endo
```

References

[1] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Proceedings of 12th ACM Symposium on Principles of Programming Languages Conference*, pages 52–66, 1985.

[2] Joseph Goguen, Claude Kirchner, and José Meseguer. *Models of Computation for the Rewrite Rule Machine.* Technical Report, SRI International, 1986.

[3] Donald Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3.* Addison-Wesley, 1973.

[4] Sany Leinwand and Joseph Goguen. *Architectural Options and Testbed Facilities for the Rewrite Rule Machine.* Technical Report, SRI International, 1986.

[5] John E. Savage. Area-Time Tradeoffs for Matrix Multiplication and Related Problems in VSLI Models. *Journal of Computer and System Sciences*, 22:230–242, 1981.

[6] Jeffrey Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1983.

END

DTIC

9-86